Korea Advanced Institute of Science and Technology

School of Electrical Engineering

EE735 Computer Vision Fall 2018

Student's Name: Dinh Vu

Student's ID: 20184187

Programming Assignment 3

The below implementation are programmed in Tensorflow 1.10.0 framework on a computer with an Intel(R) Core(TM) i7 CPU @ 4.20 GHz and a GPU NVIDIA GTX GeForce 960Ti.

1. Data Setup

CIFAR10 dataset consists 50,000 training images and 10,000 testing images. The indexes of the 4000 random labeled training images are saved in `svtrain.p` file while the unlabeled images 's indexes are stored in the `usvtrain.p` file. The CIFAR10 images are converted from binary format to numpy arrays with data type `uint8` by using the `get_data_set()` function. The corresponding labels are transformed to one-hot arrays by the `dense_to_one_hot()` function. These two functions are found in the `utils.py` file and presented in Figure 1.1 and 1.2 below.

```
21  def dense_to_one_hot(labels_dense, num_classes=10):
22    num_labels = labels_dense.shape[0]
23    index_offset = np.arange(num_labels) * num_classes
24    labels_one_hot = np.zeros((num_labels, num_classes))
25    labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
26    return labels_one_hot
```

Figure 1.1. The `dense_to_one_hot()` function

After converting CIFAR10 dataset from binary to numpy array, to speed up the training process, the data pipeline is built to read the images and their corresponding labels, then splitting them to mini-batch by using `tf.data.Dataset()` in Tensorflow.

```
32    train_x, train_y = get_data_set("train")
33    with open('..\\data\\svtrain.p', 'rb') as fp:
34      idx = pickle.load(fp)
35    train_x = train_x[idx, :]
36    train_y = train_y[idx, :]
37    num_train = train_x.shape[0]
38
39    train_data = tf.data.Dataset.from_tensor_slices((train_x, train_y))
40    train_data = train_data.shuffle(num_train)
41    train_data = train_data.map(train_parse, num_parallel_calls=8)
42    train_data = train_data.batch(batch_size)
43    train_iter = train_data.make_initializable_iterator()
44    x_train, y_train = train_iter.get_next()
```

Figure 1.3. An example of using `tf.data.Dataset()`

```python
def get_data_set(name="train"):
    x = None
    y = None

    if name is "train":
        for i in range(5):
            f = open('..\\data\\cifar-10-batches-py\\data_batch_'+str(i+1), 'rb')
            datadict = pickle.load(f, encoding='latin1')
            f.close()

            x_ = datadict["data"]
            y_ = datadict["labels"]

            x_ = np.array(x_, dtype=float) / 255.0
            x_ = x_.reshape([-1, 3, 32, 32])
            x_ = x_.transpose([0, 2, 3, 1])
            x_ = x_.reshape(-1, 32*32*3)

            if x is None:
                x = x_
                y = y_
            else:
                x = np.concatenate((x, x_), axis=0)
                y = np.concatenate((y, y_), axis=0)
    elif name is "test":
        f = open('..\\data\\cifar-10-batches-py\\test_batch', 'rb')
        datadict = pickle.load(f, encoding='latin1')
        f.close()

        x = datadict["data"]
        y = np.array(datadict["labels"])

        x = np.array(x, dtype=float) / 255.0
        x = x.reshape([-1, 3, 32, 32])
        x = x.transpose([0, 2, 3, 1])
        x = x.reshape([-1, 32*32*3])

    return x, dense_to_one_hot(y)
```

Figure 1.2. The `get_data_set()` function

2. Network Implementation

The network architecture is fixed and shown in Table 2.1 below. However, in the paper [1] the noise layer is considered as a data augmentation technique, so it can be move to the data preprocessing in the training. The softmax layer is not typed in implementation, but it will go along with the network in the prediction. Hence, generally, the structure of the given network is not changed. The network is implemented in the `model.py` file and displayed in Figure 2.1.

Table 2.1. The network architecture

NAME	DESCRIPTION
input	32×32 RGB image
noise	Additive Gaussian noise $\sigma = 0.15$
conv1a	128 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
conv1b	128 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
conv1c	128 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
pool1	Maxpool 2×2 pixels
drop1	Dropout, $p = 0.5$
conv2a	256 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
conv2b	256 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
conv2c	256 filters, 3×3, pad = 'same', LReLU ($\alpha = 0.1$)
pool2	Maxpool 2×2 pixels
drop2	Dropout, $p = 0.5$
conv3a	512 filters, 3×3, pad = 'valid', LReLU ($\alpha = 0.1$)
conv3b	256 filters, 1×1, LReLU ($\alpha = 0.1$)
conv3c	128 filters, 1×1, LReLU ($\alpha = 0.1$)
pool3	Global average pool ($6 \times 6 \rightarrow 1 \times 1$ pixels)
dense	Fully connected $128 \rightarrow 10$
output	Softmax

```
def TESnet(x, scope, drop_rate, reuse, getter=None):
  with tf.variable_scope(scope, reuse=reuse, custom_getter=getter):
    x = tf.layers.conv2d(x, 128, 3, 1, 'same', reuse=reuse, name='conv1a')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv1a/LReLU')
    x = tf.layers.conv2d(x, 128, 3, 1, 'same', reuse=reuse, name='conv1b')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv1b/LReLU')
    x = tf.layers.conv2d(x, 128, 3, 1, 'same', reuse=reuse, name='conv1c')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv1c/LReLU')

    x = tf.nn.max_pool(x, [1,2,2,1], [1,1,1,1], 'SAME', name='max_pooling_1')
    x = tf.layers.dropout(x, rate=drop_rate, name='dropout_1')

    x = tf.layers.conv2d(x, 256, 3, 1, 'same', reuse=reuse, name='conv2a')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv2a/LReLU')
    x = tf.layers.conv2d(x, 256, 3, 1, 'same', reuse=reuse, name='conv2b')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv2b/LReLU')
    x = tf.layers.conv2d(x, 256, 3, 1, 'same', reuse=reuse, name='conv2c')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv2c/LReLU')

    x = tf.nn.max_pool(x, [1,2,2,1], [1,1,1,1], 'SAME', name='max_pooling_2')
    x = tf.layers.dropout(x, rate=drop_rate, name='dropout_2')

    x = tf.layers.conv2d(x, 512, 3, 1, 'valid', reuse=reuse, name='conv3a')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv3a/LReLU')
    x = tf.layers.conv2d(x, 256, 1, 1, 'same', reuse=reuse, name='conv3b')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv3b/LReLU')
    x = tf.layers.conv2d(x, 128, 1, 1, 'same', reuse=reuse, name='conv3c')
    x = tf.nn.leaky_relu(x, alpha=0.1, name='conv3c/LReLU')

    x = tf.layers.average_pooling2d(x, 6, 1, 'same', name='avg_pool')
    x = tf.reduce_mean(x, axis=[1, 2])

    x = tf.layers.dense(x, 10)

  return x
```

Figure 2.1. The network implementation

3. Supervised Model

Before training, the images must be preprocessed. Because the supervised training data only consists 4,000 images while there are 10,000 testing images, the data augmentation is required to avoid overfitting. The training images are random flip horizontally and add random Gaussian noise. These preprocessing and data augmentation are written in the `train_parse()` function on the `utils.py` file and depicted in Figure 3.1. At the beginning of every epoch, entire training dataset is randomly shuffled.

```
72  def train_parse(x, y):
73    x = tf.reshape(x, [height, width, 3])
74    x = tf.image.random_flip_left_right(x)
75    x = tf.cast(x, tf.float32)
76    noise = tf.random_normal(tf.shape(x), mean=0.0, stddev=0.15, dtype=tf.float32)
77    x = tf.add(x, noise)
78    x = tf.clip_by_value(x, 0.0, 255.0)
79    y = tf.cast(y, tf.int32)
80    return x, y
```

Figure 3.1. The training preprocess

The Adam Optimization is used for minimize the cross entropy loss with the default parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$. In order to avoid gradient explosion caused by outliers, the gradients are clipped from -1 to 1. The model is trained with 100 epochs, batch size of 100. The initial learning rate equals to 0.001 and decayed by half after 69 epochs. The cross entropy loss and training accuracy with respect to iterations are presented in Figure 3.1 and 3.2.

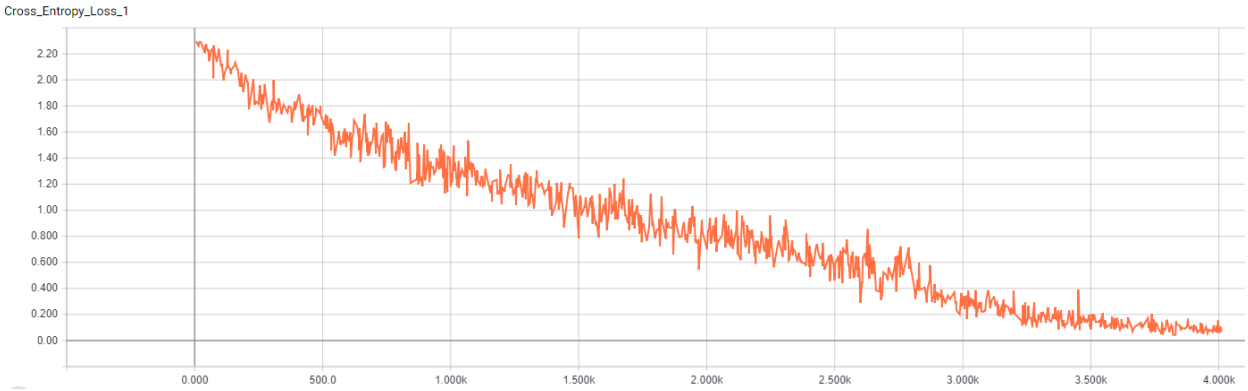

Figure 3.1. The cross entropy loss during the training process

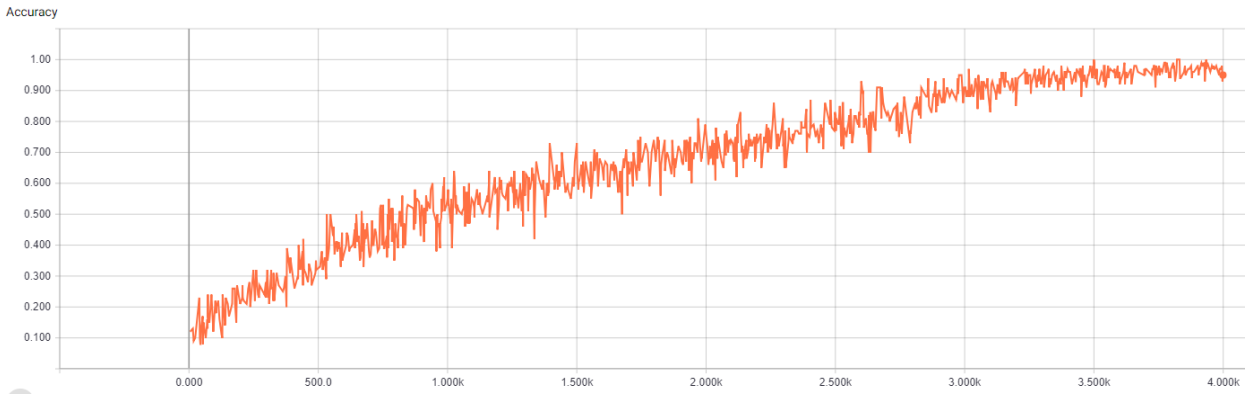

Figure 3.2. The training accuracy

The supervised training process consumed about 24 minutes. After training, the supervised model is validated with 10,000 images in CIFAR10 testing dataset and taken 75 seconds. The testing accuracy of the supervised model is 61.35 %. All supervised training process is presented in the `sv_train.py` file.

4. Mean Teacher

The training data is preprocessed same as supervised learning. The Adam optimizer is used for entire of the training process with default setup.

First, the student and teacher networks are trained with 4,000 labeled images in 120 epochs, batch size of 100, learning rate is set to 0.001 and unchanged during labeled training steps. The loss function is the summation of the classification loss and the consistency loss. Where the classification loss is the cross entropy loss in the supervised learning problem, while the consistency loss is defined as mean squared error between the output of the softmax layers in the student and teacher models

Second, the student and teacher networks are continuously trained with only the consistency loss during 60 epochs.

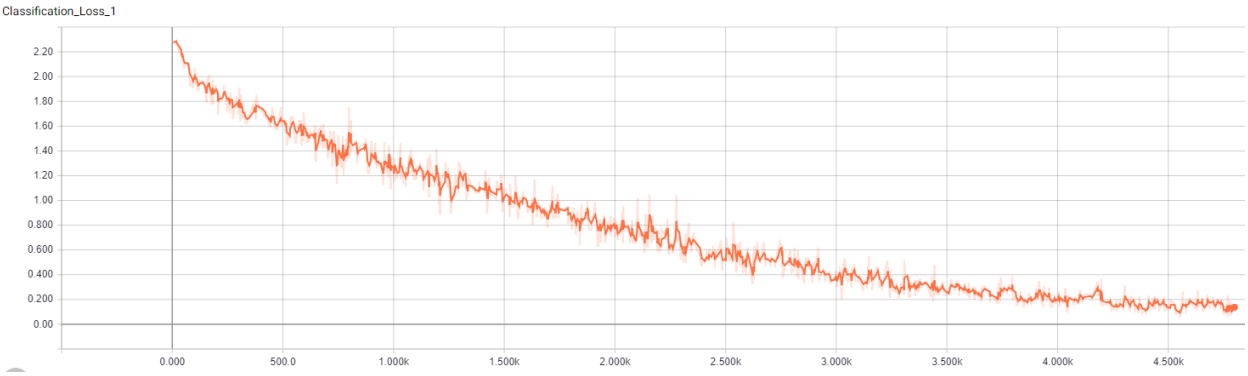

Figure 4.1. The classification loss during training with 4000 labeled samples

The testing accuracy of the student model is 57.60% and of the teacher model is 57.77%

Reference

[1] Samuli Laine and Timo Aila, “Temporal Ensembling for Semi-Supervised Learning”, *International Conference on Learning Representations*, April 2017, Toulon, France

[2] Antti Tarvainen and Harri Valpola, “Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results”, *International Conference on Learning Representations*, April 2017, Toulon, France